A: Clique

Ustalmy przedział p, i załóżmy, że to najkrótszy, jaki znajdzie się w rozwiązaniu. Wsród przedziałów z których będziemy dobierać rozwiązanie nie ma więc w szczególności takich które zawierają się wewnątrz p. Ponadto, przedziały które nie przecinają p możemy od razu pominąć, a przedziały które zawierają oba końce p (czyli albo zawierają całe p, albo zawierają oba końce i są „po drugiej stronie" okręgu) możemy bez straty ogólności wziąć.

Pozostają nam przedziały które zawierają dokładnie jeden z końców p, które nazwijmy A i B. Chcemy wybrać taki podzbiór przedziałów, że każdy wybrany przedział zawierający A przecina się z każdym wybranym zawierającym B.

Przedziały te można przekształcić na punkty na płaszczyźnie; te zawierające A zmienimy w punkty czarne, a te zawierające B - w białe. Przekształcenie wykonamy w następujący sposób.

Rozważmy przedział zawierający A, który wystaje z jednej strony tego punktu o x_a, a z drugiej o y_a, oraz inny przedział który zawiera B i wystaje z jednej strony o x_b, a z drugiej o y_b. Jeśli odległość międyz A i B z jednej strony to len_{ab}, a z drugiej to len_{ba}, to te przedziały nie przetną się jeśli $x_a + x_b < len_{ab}$ oraz $y_a + y_b < len_{ba}$. Równowaznie, $x_a < len_{ab} - x_b$ oraz $y_a < len_{ba} - y_b$. Pierwszy przedział przekształcamy na punkt czarny (x_a, y_a), zaś drugi na punkt biały $(len_{ab} - x_b, len_{ba} - y_b)$.

Po przekształceniu nasz problem redukuje się do: wybrać największy podzbiór punktów, tak że żaden biały nie jest na obu współrzędnych większy od żadnego czarnego.

Równoważnie, jeśli myślimy o *skrajnie górnym łańcuchu* wybranych białych punktów, to pod tym łańcuchem chcemy wziąć wszystkie białe punkty, a nad nim - czarne.

Obliczamy programowanie dynamiczne $dp[y]$ - wynik dla obecnie przetworzonych punktów, przy założeniu że nasz łańcuch białych punktów kończy się na wysokości co najmniej y. By to zdefiniować, wygodnie jest najpierw przemapować wszystkie współrzędne y punktów na wartości z przedziału $[0, n-1]$. Zauważmy, że dp z definicji jest nierosnące.

Przetwarzamy punkty od lewej do prawej (rosnąco po współrzędnej x), i aktualizujemy dp.

Gdy przetwarzamy punkt czarny na wysokości y, to okazuje się, że wystarczy zawiększyć o 1 wartości $dp[y']$ dla $y' \leq y$.

Gdy przetwarzamy punkt biały na wysokości y, to chcemy zawiększyć o 1 wartości $dp[y']$ dla $y' \geq y$. Z drugiej strony musimy też zachować monotoniczność dp. Otrzymujemy, że wystarczy dodać 1 na prefiksie potencjalnie dłuższym niż ten zaczyanjący się na pozycji y; dokładniej, zaczynającym się w miejscu, gdzie wartość dp spada do poziomu $dp[y]$, co staje się na pozycji y lub wcześniej.

Na koniec nasz wynik znajduje się w $dp[0]$.

Pozostaje stwierdzić jakiej struktury danych używamy do reprezentacji dp. Potrzebujemy dodawać 1 na przedziale, odczytywać wartość w punkcie, i znajdować dla zadanego v najmniejsze i takie że $dp[i] \leq v$. Do tego wystarcza proste drzewo przedziałowe: w każdym węźle trzymamy wartość dodaną na przedziale bazowym i jego minimum. Dzięki minimum jesteśmy w stanie znajdować szukaną pozycję i w $O(\log n)$ schodząc od góry drzewa.

Podsumowując, rozwiązanie podproblemu o białych i czarnych punktach zajmuje czas $\mathcal{O}(n \log n)$. Jako że wykonujemy to n razy, to finalna złożoność to $\mathcal{O}(n^2 \log n)$.

B: Common subsequence

Rozwiązania wzorcowe działają dla dowolnego warunku, który określa czy element q może występować po p w szukanym podciągu. Tak więc używając dokładnie tego samego algorytmu, zamiast podciągu spełniającego warunek podany w zadaniu, moglibyśmy też np szukać najdłuższego wspólnego podciągu rosnącego, albo podciągu o każdych dwóch sąsiednich wyrazach względnie pierwszych.

Podejście 1:
Obliczamy dynamicznie $dp[x][y]$ – najdłuższy wspólny podciąg $A[1..x]$ i $B[1..y]$ kończący się symbolem $B[y]$. Jeśli to się nam uda, to największa z wartości $dp[\cdot][\cdot]$ będzie rozwiązaniem, a odtworzyć ciąg jest łatwo przechowując wskaźniki wsteczne. Załóżmy, że obliczyliśmy cały wiersz $dp[x-1][\cdot]$ i przechodzimy do wiersza $dp[x][\cdot]$. Zauważmy, że między tymi wierszami mogą zmienić się tylko te wartości, dla których $A[x] = B[y]$, jeśli bowiem $A[x] \neq B[y]$, to $A[x]$ nie może wystąpić w ciągu kończącym się na $B[y]$. Niech więc $\alpha = A[x]$, i spróbujmy poprawić te wartości, dla których $B[y] = \alpha$. Wiemy, że ciąg kończący się na α musi mieć jakiś poprzedni element $\alpha\prime$, który występuje gdzieś w ciągu B i różni się od α na nie więcej niż k bitach, co zapisujemy $\alpha \approx \alpha\prime$. A zatem jeśli nasz ciąg ma się kończyć na $A[x] = B[y]$, to $dp[x][y] = 1 + \max_{j<y, B[j]\approx\alpha} dp[x-1][j]$. Możemy zatem jedną pętlą, dla każdego $z = 1, 2 \ldots, m$ policzyć kolejno $P[z] = \max_{j<z, B[j]\approx\alpha} dp[x-1][j]$. Teraz już widać, że $dp[x][y] = \max(dp[x-1][y], P[y])$.

Podejście 2:
Powyższy dynamik $dp[x][y]$ możemy też policzyć w trochę inny sposób. Nieformalnie: $dp[x][y]$ oznacza że rozważany podciąg kończy się na $B[y]$, spróbujmy przejścia wykonywać w ten sposób żeby albo x albo y zawsze pamiętał co zaplanowaliśmy jako ostatni element naszego ciągu. Jeśli tym elementem jest $B[y]$, to możemy dodać przejścia dynamika zmniejszające x; jeśli w którymś momencie $A[x] = B[y]$ to możemy zapamiętać (jako dodatkowy bit stanu dynamika) że element który wybraliśmy w B już widzieliśmy w A. Jeśli zaś $A[x]$ jest elementem który może występować przed $B[y]$, to możliwym przejściem jest wybrać go jako przedostatni element naszego ciągu. Teraz możemy dodać 1 do wyniku, i zacząć przesuwać y, trzymając x żeby pamiętać ostatni element.

Dokładniej, nasz dynamik będzie miał teraz wymiar $dp[n][m][2][2]$. Pierwszy z dodatkowych bitów pamięta, czy ostatni element rozważanego ciągu to $A[x]$ (jeśli bit jest 0) czy $B[y]$ (jeśli 1). Drugi bit pamięta, czy ustalony przez nas ostatni element ciągu już widzieliśmy w drugim z ciągów.

Przy tej definicji dynamika nie potrzebujemy specjalnej kolejności odwiedzania stanów: stan $dp[x][y][b_1][b_2]$ obliczymy na podstawie stanów postaci $dp[x-1][y][\cdot][\cdot]$ i $dp[x][y-1][\cdot][\cdot]$.

C: Contamination

Zadanie polega na tym, żeby dla ustalonego zbioru kół (domkniętych) na płaszczyźnie odpowiadać na zapytania postaci (p, q, y_{min}, y_{max}), gdzie $p = (p_x, p_y)$ i $q = (q_x, q_y)$ to dwa punkty z których żaden nie leży w którymś z kół, a y_{min} i y_{max} to dwie liczby takie, że $y_{min} \leq y_{max}$. Dla każdego takiego zapytania należy stwierdzić, czy istnieje krzywa o końcach w p i q, która w całości zawiera się w pasie $Y = \{(x, y) \in R^2 \mid y_{min} \leq y \leq y_{max}\}$.

Łatwo się przekonać, że taka krzywa istnieje wtedy i tylko wtedy, gdy nie istnieje koło, którego pionowa średnica "oddziela" punkty p i q w pasie Y, to znaczy gdy nie istnieje takie koło o środku (c_x, c_y) i promieniu r, że $\min(p_x, q_x) \leq x \leq \max(p_x, q_x)$ oraz $c_y - r \leq y_{min} \leq y_{max} \leq c_y + r$.

Problem ten możemy rozwiązać techniką zamiatania płaszczyzny poziomą miotłą od najmniejszych współrzędnych y do najwyższych. W każdym momencie utrzymujemy na drzewie przedziałowym zbiór aktywnych pionowych średnic. Pionowa średnica koła o środku (c_x, c_y) i promieniu r staje się aktywna na wysokości $c_y - r$, a przestaje być aktywna na wysokości $c_y + r$. Odpowiedź na zapytanie (p, q, y_{min}, y_{max}) znajdujemy na wysokości y_{min}, po prostu sprawdzając na drzewie przedziałowym maksimum na przedziale $[\min(p_x, q_x), \max(p_x, q_x)]$. Jeśli to maksimum wynosi co najmniej y_{max}, to odpowiedź na zapytanie to "NIE", a w przeciwnym razie "TAK". Złożoność powyższego algorytmu to $\mathcal{O}(n \log n)$

D: Chance

Niech r - rząd elemena x w grupie $Z^*_{n!}$. Wtedy $lim_{k\to\infty}\ p(k) = lim_{k\to\infty} \frac{\lfloor \frac{k}{r} \rfloor}{k} = \frac{1}{r}$. Czyli tak naprawdę chcemy znaleźć tę liczbę r. Niech $n! = p_1^{\alpha_1} \dots p_k^{\alpha_k}$ – rozkład $n!$ na liczby pierwsze. Wtedy jeśli $\forall_i\ x^j \equiv 1\ (mod\ p^k)$ to $x^j \equiv 1\ (mod\ n!)$. Tak naprawdę chcemy dla każdego $1 \leq i \leq k$ znaleźć r_i – minimalną liczbę naturalną dodatnią taką, że $x^{r_i} \equiv 1\ (mod\ p_i^{\alpha_i})$. Wtedy $r = lcm(r_1, r_2, \dots, r_k)$.

Ale jak obliczyć r_i? Trzeba znaleźć rząd elementu x w grupie $Z^*_{p_i^{\alpha_i}}$? Zauważmy, że $Z^*_{p_i^{\alpha_i}} \cong Z_{\phi(p_i^{\alpha_i})} = Z_{p_i^{\alpha_i - 1} \cdot (p_i - 1)}$. Na mocy twierdzenia Lagrange'a wiemy, że $r_i | p_i^{\alpha_i - 1} \cdot (p_i - 1)$. Ponieważ p_i jest małe, możemy przeiterować się po wszystkich dzielnikach liczby $p_i^{\alpha_i - 1} \cdot (p_i - 1)$ (zauważmy, że ta liczba może być duża, ale rozkład takiej liczby jest prosty, ponieważ znamy p_i i α_i) i znaleźć najmniejszą taką, że $x^{r_i} \equiv 1\ (mod\ p_i^{\alpha_i})$.

Haczykiem w tym zadaniu było to, że odpowiedź może być duża i nie mieści się w typie *long long*, dlatego należy zaimplementować własną arytmetykę albo skorzystać z wbudowanego typu *__int128*, który nie jest w stardardzie *C++*, ale jest dostępny na dużej części sprawdzaczek, w tym na Satori.

E: Coloring

Żeby graf o n wierzchołkach pokolorować krawędziowo n kolorami, wystarczy krawędź łączącą wierzchołek i z wierzchołkiem j pokolorować kolorem $(i + j) \bmod n$.

Naturalna implementacja tego rozwiązania na przemian wczytuje po jednej krawędzi i wypisuje jej kolor. Jeśli każda operacja wejścia będzie opróżniać bufor wyjścia (domyślne zachowanie strumieni `cin`/`cout` w C++), całość będzie działać bardzo wolno (ok. 10 razy wolniej od rozwiązania wzorcowego). Żeby uniknąć tego problemu, należy wyłączyć związanie strumienia wejściowego i wyjściowego, wywołując na początku programu `cin.tie(nullptr)`. Alternatywnie, można użyć biblioteki `stdio` lub najpierw wczytać całe wejście, zapisując je w tablicy, a potem wypisać całe wyjście.

F: Chillout

Podejście 1:
Jeśli Vladik zaprosi przyjaciela z wymaganiem d_i, to bez straty ogólności może też zaprosić wszystkich z wymaganiem nie większym niż d_i. W związku z tym, jeśli posortujemy przyjaciół po d_i, to jedyne rozwiązania jakie opłaca się sprawdzić to prefiksy tej posortowanej kolejności. Wystarczy więc przeiterować się po prefiksach posortowanej kolejności utrzymując maksymalny i minimalny rating. Złożonosć: $\mathcal{O}(n \log n)$.

Podejście 2:
Możemy posortować przyjaciół po ratingu r_i. Niech najmniejszy i największy rating to odpowiednio m i M. Jeśli osoby o najminejszym i największym ratingu mają wymaganie d_i nie większe niż $M-m$, to bez straty ogólności możemy wziąć te dwie osoby, i wszystkie inne o wymaganiu nie większym niż $M-m$. Jeśli któraś ze skrajnych osób ma za wysokie wymaganie, to na pewno nie będziemy w stanie jej zaprosić. Można więc usunąć ją z rozważań i kontynuować obgryzanie skrajnych osób. Złożonosć tego podejścia to również $\mathcal{O}(n \log n)$.

G: Coffee

O co tak naprawdę chodziło w tym zadaniu: wyszukujemy pewną liczbę $x \in (a, b]$; w jednym pytaniu możemy zapłacić y złotych i dowiedzieć się, czy $x \leq y$. I teraz zastanawiamy się, ile minimalnie potrzebujemy wydać pieniędzy w najgorszym możliwym przypadku, żeby znaleźć tego głupiego x-a. Ponieważ $b \leq 3000$ od razu nasuwa się następujące rozwiązanie: będziemy liczyć dynamika $f(a, b)$ – minimalny koszt znalezienia x, jeśli wiemy, że $x \in (a, b]$. Super, ale jak wyliczać taką funkcję? Przeiterujmy się po wszystkich możliwych pierwszych zapytaniach (w oczywisty sposób zapytanie też leży w przedziale $(a, b]$). Wtedy nasz oryginalny przedział $(a, b]$ rozpada się na dwa mniejsze, dla których już wyliczyliśmy odpowiedź. Inaczej mówiąc, chcemy obliczyć taką funkcję:

$$f(a,b) = \begin{cases} 0 & b-a=1 \\ \min_{a<c<b}\{\max\{f(a,c), f(c,b)\} + c\} & b-a>1 \end{cases}$$

Niech $n := b - a$. Wtedy powyższe rozwiązanie ma złożoność $\mathcal{O}(n^3)$, co jest trochę za wolno. Celujemy w $\mathcal{O}(n^2)$, ewentualnie w $\mathcal{O}(n^2 \log n)$ z małą stałą. Wąskim gardłem naszego algorytmu jest to, że obliczanie $f(a, b)$ dla ustalonych a, b tra liniowo.

Obserwacja 1. *Ciąg $f(a, a+1), f(a, a+2), \ldots, f(a, b)$ jest niemalejący.*

Obserwacja 2. *Ciąg $f(a, b), f(a+1, b), \ldots, f(b-1, b)$ jest nierosnący.*

Dowód. Te dwie obserwacje wynikają stąd, że jeśli musimy zapłacić r złotych, żeby znaleźć x na pewnym przedziale $(a, b]$, to znalezienie x-a na przedziale $(a+1, b]$ czy $(a, b-1]$ nie będzie kosztować więcej, niż r złotych. □

Niech $p(a, b) := \min_{a \leq c \leq b}\{f(a, c) \geq f(c, b)\}$. Zauważmy, że p jest dobrze zdefiniowaną funkcją. Co więcej, możemy zastosować wyszukiwanie binarne, żeby w czasie $\mathcal{O}(\log(b-a))$ znaleźć $p(a, b)$. Wtedy możemy przepisać wzorek na f następująco:

$$f(a,b) = \min\{\min_{a<c<p(a,b)}\{f(c,b)+c\}, \min_{p(a,b)\leq c<b}\{f(a,c)+c\}\}$$

Zdefiniujmy sobie $g_1(a,b) := f(a,b) + a$ i $g_2(a,b) := f(a,b) + b$. Wtedy

$$f(a,b) = \min\{\min_{a<c<p(a,b)}\{g_1(c,b)\}, \min_{p(a,b)\leq c<b}\{g_2(a,c)\}\}$$

Czyli chcielibyśmy szybko umieć znajdować minimum funkcji na przedziale dla $g_1(\cdot,b)$ i $g_2(a,\cdot)$. No i widać, czym to pachnie (albo śmierdzi, kto jak woli). Będziemy trzymać drzewa przedziałowe dla $g_1(\cdot,b)$ i $g_2(a,\cdot)$. I po obliczeniu kolejnego $f(a,b)$ będziemy aktualizować odpowiednie dwa drzewa przedziałowe.

W taki sposób dostaliśmy rozwiązanie w złożoności $\mathcal{O}(n^2 \log n)$, które ma dużą stałą. Taki rozwiązanie może wejść, a może i nie wejść xD.

Kilka hintów, jak zbić złożoność do $\mathcal{O}(n^2)$.

- dla funkcji $p(a,b)$ zachodzi następująca własność: $p(a,b-1) \leq p(a,b) \leq p(a+1,b)$ (te nierówności wynikają z obserwacji powyżej).
- Ponieważ zapytania dla $g_1(\cdot,b)$ i $g_2(a,\cdot)$ przesuwają się w jedną stronę (w prawo dla $g_1(\cdot,b)$ i w lewo dla $g_2(a,\cdot)$) to zamiast drzew przedziałowych dla g_1 i g_2 możemy trzymać kolejki, które pozwalają sprawdzać minimalną wartość w tej kolejce w czasie $\mathcal{O}(1)$.

Na koniec dodamy, że w tym zadaniu mieliśmy bardzo dużo zapytań o różne przedziały $(a,b]$. Ale okazuje się, że to nie jest żaden problem, ponieważ możemy wyliczyć funkcję f tylko raz na samym początku i potem wykorzystywać stablicowane wartości, żeby odpowiadać na zapytania.